1. Create an assert statement that throws an AssertionError if the variable spam is a negative integer.

assert spam >= 0, "spam should not be a negative integer"

2. Write an assert statement that triggers an AssertionError if the variables eggs and bacon contain strings that are the same as each other, even if their cases are different (that is, 'hello' and 'hello' are considered the same, and 'goodbye' and 'GOODbye' are also considered the same).

assert eggs.lower() != bacon.lower(), "eggs and bacon should not be the same"

3. Create an assert statement that throws an AssertionError every time.

assert False, "This assertion always fails"

4. What are the two lines that must be present in your software in order to call logging.debug()?

import logging

logging.basicConfig(level=logging.DEBUG)

The first line imports the logging module, which provides the logging functionality in Python. The second line configures the logging system to enable debug-level logging. It sets the root logger's level to DEBUG, which allows debug-level log messages to be output.

With these two lines in place, you can call logging.debug() to log debug-level messages in your code.

5. What are the two lines that your program must have in order to have logging.debug() send a logging message to a file named programLog.txt?

import logging

logging.basicConfig(filename = ‘programLog.txt’ ,level=logging.DEBUG)

6. What are the five levels of logging?

The five levels of logging, in increasing order of severity, are:

1. DEBUG: Detailed information, typically used for debugging purposes.
2. INFO: General information about the program's execution.
3. WARNING: Indicates a potential issue or something that may cause problems in the future, but the program can still continue running.
4. ERROR: Indicates an error occurred, but it is still possible for the program to continue running.
5. CRITICAL: Indicates a critical error or a severe issue that prevents the program from continuing to run.

These levels allow you to categorize and prioritize the log messages based on their importance and severity. By setting the logging level, you can control which messages are recorded based on the severity level you want to capture.

7. What line of code would you add to your software to disable all logging messages?

logging.disable(logging.CRITICAL)

8.Why is using logging messages better than using print() to display the same message?

Using logging messages is generally considered better than using print() for several reasons:

1. **Logging levels**: The logging module provides different levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL) to categorize log messages based on their severity. This allows you to control which messages are displayed or recorded based on the desired level of detail. With print(), you don't have built-in levels, so it's harder to manage the verbosity of your program.
2. **Configurability**: The logging module allows you to configure the behavior of logging messages, such as the output destination (e.g., console, file), formatting, and filtering based on levels or other criteria. This gives you more flexibility and control over how logging information is handled. With print(), you have limited control over the output format and destination.
3. **Granular control**: Logging messages can be selectively enabled or disabled based on their severity levels or other criteria. This means you can include more detailed logging statements during development and debugging, and easily disable them in production to improve performance. With print(), you would need to manually remove or comment out the statements to disable them.
4. **Integration with existing logging infrastructure**: The logging module integrates well with other logging systems and frameworks. It allows you to leverage features such as log rotation, log aggregation, and integration with third-party logging services. Using print() would require custom implementation for such functionalities.
5. **Standardization and consistency**: By using the logging module, you follow a standardized approach to logging, making your code more consistent and maintainable. It also allows other developers to easily understand and work with your logging statements, as they are familiar with the logging module's API.

Overall, using the logging module provides more flexibility, configurability, and control over logging messages compared to using print(), making it a preferred choice for logging in Python applications.

9. What are the differences between the Step Over, Step In, and Step Out buttons in the debugger?

The Step Over, Step In, and Step Out buttons are commonly found in debuggers and are used to control the execution flow while debugging. Here are the differences between these buttons:

1. **Step Over**: When you click the Step Over button, the debugger executes the current line of code and moves to the next line. If the current line contains a function call, the debugger will not enter the function and will instead move to the next line in the current function. It allows you to skip over the details of the function and focus on the higher-level execution.
2. **Step In**: Clicking the Step In button will cause the debugger to move to the next line of code, just like Step Over. However, if the current line contains a function call, the debugger will enter the called function and move to the first line of that function. It allows you to dive into the details of the called function and debug it line by line.
3. **Step Out**: The Step Out button is used when you are inside a function and want to quickly execute the remaining lines of that function without stepping through each line. When you click Step Out, the debugger will continue execution until it reaches the line that returns from the current function and then stops. It allows you to quickly move out of the current function and return to the calling code.

In summary, Step Over is used to execute the current line and move to the next line, Step In is used to enter a function and move to its first line, and Step Out is used to quickly execute the remaining lines of the current function and return to the calling code. These buttons provide control over the execution flow during debugging and help in understanding and troubleshooting the code.

10.After you click Continue, when will the debugger stop ?

After clicking Continue in a debugger, the debugger will stop when one of the following conditions is met:

1. A breakpoint is encountered: If you have set breakpoints at specific lines in your code, the debugger will stop when it reaches any of those breakpoints. Breakpoints are set to pause the execution and allow you to inspect the program's state at that point.
2. An exception is raised: If an unhandled exception occurs during the execution of your program, the debugger will automatically stop and display the traceback and error information. This allows you to examine the state of the program at the point of the exception.
3. The program's execution is completed: If the program runs to its end without encountering any breakpoints or exceptions, the debugger will stop once the program's execution is completed. At this point, you can review any captured logging or debugging information.

It's important to note that clicking Continue essentially instructs the debugger to let the program continue its normal execution until one of the stopping conditions is met. The debugger will not pause or interrupt the program's flow unless one of these conditions occurs.

11. What is the concept of a breakpoint?

The concept of a breakpoint is a feature in a debugger that allows you to pause the execution of a program at a specific line of code or at a specific condition. When a breakpoint is encountered during the execution of the program, the debugger will pause the program's execution, allowing you to inspect and analyze the program's state at that point.

Breakpoints are useful for debugging and troubleshooting purposes. They enable you to examine variables, check the program's flow, and step through the code line by line to identify and understand any issues or unexpected behavior in the program.

By setting breakpoints strategically in your code, you can gain more insight into the program's behavior and investigate specific areas of interest. Breakpoints help you narrow down the scope of debugging and focus on specific portions of the code where you suspect the problem may lie.

Once a breakpoint is reached, you can use the debugger's features to examine variables, step through the code, modify variables for testing purposes, and gather information to diagnose and fix any issues in your program.